ctypes. ctypes run!

Alex Holkner

This paper was originally presented at the Open Source Developer's Conference, which ran 5-8 December, 2006 in Melbourne, Australia. It was reviewed at that time, and selected for publication here as being of an excellent standard.

One of the new features of Python 2.5 is the introduction of ctypes as a standard library module. At the simplest level, ctypes adds the standard C types to Python: signed and unsigned bytes, shorts, ints and longs; as well as structs, unions, pointers and functions. At run-time it can load a shared library (DLL) and import its symbols, allowing a Python application to make function calls into the library without any special preparation. ctypes can be used to wrap native libraries in place of interface generators such as SWIG, to manipulate memory and Python objects at the lowest level, and to prototype application development in other languages.

This paper begins with a quick introduction to ctypes, shows some advanced techniques, and describes some examples of how it has been used by the author in his recent work.

Introduction to ctypes

Using C types in Python

ctypes is standard with Python 2.5, and can be installed as an extension module for Python 2.4 and earlier (see the references at the end of this paper for the download site).

Importing ctypes gives you access to the standard C types that are not available in Python.

```
>>> from ctypes import *
>>> i = c_ubyte(255)          Create an unsigned byte with initial value 255.
>>> i
c_ubyte(255)
>>> i.value                   The "value" attribute gives an int, and can be modified.
255
>>> i.value += 1
>>> i.value
0                             Attempting to set the byte to 256 wraps the value to 0
```

Notice that the value of a c_ubyte is limited to the range [0, 255], and the overflow behaviour is identical to that in C. The ranges of these types depend on your platform, but ctypes also provides known-width datatypes c_int8, c_int16, and so on. The simple numeric types are shown below:

C type	*Unsigned*	*Signed*
char	c_ubyte	c_char, c_byte
short	c_ushort	c_short
int	c_uint	c_int
long	c_ulong	c_long
long long	c_ulonglong	c_longlong
float		c_float
double		c_double

You can use c_void_p as a void pointer type, and c_char_p as a char pointer type. ctypes interprets a c_char as a byte representing a Python string of length 1, and c_char_p as an NULL-terminated string.

Arrays can be constructed by first defining the array type and then instantiating:

```
>>> my_array_type = c_int * 5              Define an int array type of length 5.
>>> array = my_array_type(1, 2, 3, 4, 5)   Create an actual array with some initial data.
>>> array
<__main__.c_int_Array_5 object at 0x2aaaaab582d0>
>>> list(array)
[1, 2, 3, 4, 5]                            The contents of the array can be seen by making a list.
```

A pointer type can be created for any basic type:

```
>>> i = c_int(42)                          Create an int with initial value 42.
>>> my_pointer_type = POINTER(c_int)       Create a type int*.
>>> ptr = my_pointer_type(i)               Create an int* pointer to i.
>>> ptr
<ctypes.LP_c_int object at 0x2aaaaab58450>
>>> ptr.contents
c_int(42)                                  The "contents" attribute of the pointer resolves indirection.
```

As a shortcut, you can create a pointer to an instance without creating the type first:

```
>>> i = c_int(42)
>>> ptr = pointer(i)
```

Unlike, C, ctypes is strict about passing arrays of different lengths and pointers to functions that expect otherwise. For example, ordinarily you cannot pass an array to a function expecting a pointer. You can make explicit casts between pointers and arrays when you need to:

```
>>> a = (c_int * 5)()                      Create an array of int of size 5.
>>> list(a)
[0, 0, 0, 0, 0]                            The array is initialized to zeros by default.
>>> ptr = cast(a, POINTER(c_int))          Cast the array to a pointer to int.
>>> ptr.contents
c_int(0)                                   The contents of the pointer is the first element of the array.
>>> ptr.contents.value = 13
>>> list(a)
[13, 0, 0, 0, 0]                           Changing pointer's contents affects the original array.
```

Defining a struct type is done by subclassing Structure:

```
>>> class Point(Structure):                This struct is equivalent in C to:
...     _fields_ = [                           typedef struct {
...       ('x', c_short),                          short x;
...       ('y', c_short)                           short y;
...     ]                                      } Point;
...
>>> sizeof(Point)
4
>>> p = Point()
>>> p.x
0
```

The struct will follow the de-facto rules for packing as in C, but member alignment can be overridden if necessary. Union types can be defined in a similar way, and structs and unions can be nested, even anonymously.

Calling library functions from Python

ctypes can read symbols and function entry points from Windows DLLs, Unix shared libraries and OS X Mach-O objects and frameworks. You can directly specify the name of a library to load, but it is more platform neutral to let ctypes locate it for you:

```
>>> from ctypes.util import find_library
>>> find_library('SDL')                    SDL is an open-source cross-platform media library.
'libSDL-1.2.so.0'                          On Linux.
'/Library/Frameworks/SDL.Framework/SDL'    On OS X.
'C:\\WINDOWS\\system32\\SDL.dll'           On Windows.
```

Once you have obtained the name of the library, you can load it. Symbols will be imported on demand. For example, on Unix systems we can load libc to get the standard C functions:

```
>>> libc = cdll.LoadLibrary(find_library('c'))      find_library returns 'libc.so.6' on Linux.
>>> libc.strcmp('alice', 'bob')
-1
```

Notice that ctypes automatically converts Python string objects to c_char_p, and assumes the return value of the function is c_int, which it converts to a Python int. You can explicitly tell ctypes the return type and argument types of a function by setting the restype and argtypes attributes:

```
>>> libc.strcmp.restype = c_int
>>> libc.strcmp.argtypes = [c_char_p, c_char_p]
```

In this case the only benefit is that ctypes will now raise an exception if you pass incorrect arguments to strcmp. For other functions, however, it is often necessary to describe the argument and return types.

On Windows, the system libraries are readily available and can be used directly:

```
>>> windll.kernel32.GetTickCount()
469494577
```

Python callback functions

ctypes allows Python functions or bound methods to be used as callbacks from library functions. To do this, create a type for the callback function specifying its arguments and return type, then create a callback by instantiating this type with your function. This example uses the standard C qsort (quicksort) function, using a Python function as the comparator:

```
>>> compare_type = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> def compare(ptr_a, ptr_b):
...     return cmp(ptr_a.contents.value, ptr_b.contents.value)
...
>>> libc.qsort.restype = None
>>> libc.qsort.argtypes = [c_void_p, c_size_t, c_size_t, compar_type]
>>> data = (c_int * 5)(5, 2, 3, 1, 4)
>>> list(data)
[5, 2, 3, 1, 4]
>>> libc.qsort(data, len(data), sizeof(c_int), compare_type(compare))
>>> list(data)
[1, 2, 3, 4, 5]
```

You must take care to keep a reference to the function type instance alive for as long as the library will use it—neglecting this was the most common error made by the author in recent work.

Advanced uses of ctypes

Pointer arithmetic

Pointer arithmetic can be useful, for example, for passing just part of a large array to a library function. This is not supported in ctypes, in that addition is not overloaded and the internal

address for a pointer is not directly modifiable. There are several ways around this, however. One way is to use the from_address method, which is used to construct a ctype instance from a given memory address:

```
>>> def ptr_add(ptr, offset):
...     address = addressof(ptr.contents) + offset
...     return pointer(type(ptr.contents).from_address(address))
...
>>> data = (c_int * 5)(1, 2, 3, 4, 5)
>>> ptr = cast(data, POINTER(c_int))
>>> ptr.contents
c_int(1)
>>> ptr = ptr_add(ptr, 3 * sizeof(c_int))
>>> ptr.contents
c_int(4)
```

Manipulating CPython objects

In the standard implementation of Python, all objects are represented internally with a struct that extends PyObject. Manipulating custom types introduced by extension libraries is possible by creating a ctypes type to match the extension type. Such a type should begin with the equivalent of PyObject_HEAD. For example, the Numeric struct is given as:

```
class _Numeric_PyArrayObject(Structure):
    _fields_ = [('ob_refcnt', c_int),
                ('ob_type', c_void_p),
                ('data', c_void_p),
                ('nd', c_int),
                ('dimensions', POINTER(c_int)),
                ('strides', POINTER(c_int)),
                ('base', c_void_p),
                ('descr', c_void_p),
                ('flags', c_uint),
                ('weakreflist', c_void_p)]
```

You can't instantiate this type directly from a Numeric array, but fortunately the id() function returns the address of any object (this is not documented behaviour). Now you can directly manipulate fields of the Numeric array, such as its base pointer, which is otherwise not possible.

```
>>> import Numeric
>>> data = Numeric.array([1, 2, 3, 4], Numeric.Int32)
>>> data_obj = _Numeric_PyArrayObject.from_address(id(data))
>>> libc.free(data_obj.data)
>>> libc.calloc.restype = c_void_p
>>> data_obj.data = libc.calloc(10, sizeof(c_int32))
>>> data_obj.dimensions.contents.value = 10
>>> print data
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0],'i')
```

In this example, the base pointer of a Numeric array is changed to a buffer we create ourselves. This is fairly contrived, in that it does nothing that you couldn't do just by creating a new array, but it does show how to overlay a Numeric array over any preexisting data, for example, a buffer returned by another library—something not possible with Numeric alone.

Automatic wrapper generation

ctypes_codegen is an experimental utility available from the ctypes project page. It will read one or more header files for a library and write a Python module that includes all the necessary structs, unions, constants and function prototypes for direct use.

Examples

Prototype application development

ctypes makes it very practical to develop your C applications by prototyping. Instead of beginning with the main() function and all the required utility of parsing command-line arguments, reading data files and formatting output, start instead with the critical algorithms, and write support code in Python.

```
int complex_algorithm(int a, int b)
{
    return a * b;
}
```

```
gcc -fpic -shared -o alg.so alg.c
```

```
>>> alg = cdll.LoadLibrary('./alg.so')
>>> alg.complex_algorithm(6, 7)
42
```

Even if the final product will not use Python, the programming overhead of using ctypes is so low that the advantages of rapid development and interactive debugging will make it easily worthwhile.

Pygame and SDL

Pygame is a Python extension that provides graphics, audio and input functions for creating games and multimedia applications. Internally it uses SDL for cross-platform media support.

Earlier this year the author reimplemented Pygame as a pure Python module using ctypes as a project sponsored by Google Summer of Code. The project began by writing ctypes definitions for all of the SDL structs and functions, and the Pygame functionality was rewritten over the top of this layer. Automatic code generation was not used, as there were many cases in which a more “Pythonic” API was required than the C one.

Performance was a major concern in this project, as image processing is something that Python generally does slowly. For simple processing and conversion, regular expressions are used on the image buffer, which run surprisingly fast – comparable to native C code. For more advanced processing the Python Imaging Library (PIL) was used. In the future it is envisioned that many of these functions will have multiple implementations making use of pure Python, PIL, native code and PyRex.

The Long Term

The C implementation of Python, now at version 2.5, is becoming just one in a community of Python implementations. Jython (for Java), IronPython (for the .NET platform) and PyPy (a statically and JIT compiling monster) are all progressing quickly and are becoming ready for production use. Standard Python extensions, including those generated by SWIG, cannot be used with these implementations, but they are all expected to support ctypes in the future if they do not already. When this is the case, native code can be used by any implementation of Python with a single ctypes wrapper module.

For this reason, as well as the simplification of development and maintenance, large projects are being reimplemented using ctypes. SDL and Pygame have been discussed, and recent work on OpenGL-ctypes is nearing completion.

References

ctypes project page

http://starship.python.net/crew/theller/ctypes/

Google Summer of Code
http://code.google.com/soc/

Pygame project page
http://www.pygame.org/

Pygame-ctypes and SDL-ctypes project page
http://www.pygame.org/ctypes

OpenGL-ctypes project page
http://pyopengl.sourceforge.net/ctypes/